(2)

RSRE
MEMORANDUM No. 4540

ROYAL SIGNALS & RADAR ESTABLISHMENT

A FRONT END IMPLEMENTATION OF THE POLLYANNA SECURE DBMS

Author: G R Hutchinson

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

RSRE MEMORANDUM No. 4540

Royal Signals and Radar Establishment

Memorandum 4540

Title: A Front End Implementation of the Pollyanna Secure DBMS.

Author: G.R.Hutchinson

Date: November 1991

ABSTRACT

Pollyanna is a simple secure DBMS which utilises polyinstantiation. It has been designed and implemented to illustrate the problems which may be encountered when applications are built on top of a polyinstantiating DBMS. The implementation comprises of a front end to a standard commercial RDBMS. The front end performs query modification and result filtering to achieve security. This document describes Pollyanna, the front end software and some of the problems which arose during the design of the system.

Contents

1. INTRODUCTION

When military, multi-level security is required in a database, confidentiality controls must be enforced over and above those controls normally implemented in a commercially available Database Management System (DBMS). One approach, called polyinstantiation [1], has been widely adopted, but provides confidentiality at the expense of integrity [4]. Pollyanna[1] is a Relational DBMS (RDBMS) which has been developed to illustrate these problems.

Pollyanna has been implemented by imposing a software front end on the standard commercially available Oracle RDBMS, a technique discussed in [3]. Oracle is used to store all the data and classifications and to perform query processing. The front end modifies the users' queries, by adding various checks, and filters the results to ensure that the users do not obtain information for which they do not have a clearance.

The Pollyanna front end was based on a front end implementation of the SWORD secure DBMS [2]. SWORD does not use polyinstantiation to achieve security but it was possible to reuse much of the software.

The front end provides an interactive query processor. This allows the user to enter queries, written in a subset of SQL, and returns any results by displaying them on the screen. The software is written C with some embedded SQL. The software runs on VAX/VMS and Apple Macintosh.

2. THE POLLYANNA SECURE RDBMS

2.1 Classifications and Clearances

Pollyanna supports classifications which comprise of three parts: Hierarchies, Caveats and Categories. The hierarchical components are typically 'Unclassified', 'Confidential', 'Secret' and 'Top secret'. Caveats typically 'UK eyes only' or 'Nato' while categories are typically 'Nuclear', 'Crypto' and other code words.

Classification A dominates classification B if:
	the hierarchical component of A is greater than that of B
and	all the caveats of A are included in those of B
and	all the categories of B are included in those of A

Each user of the system is assigned a classification which is their clearance. When a user logs onto the Pollyanna system, they select a session level which must be dominated by their clearance.

The Pollyanna system enforces a security policy of "no flows down". That is no user can observe information entered by a user with a higher clearance.

[illegible] Labelling in Pollyanna

[illegible] it is possible to classify data at the individual field level [1], all secure DBMS products based on polyinstantiation, that have been announced to date, only provide row level labelling. Since Pollyanna has been developed to illustrate problems with mounting applications on these DBMSs, it too only provides row level labelling.

Pollyanna actually applies classifications at both the table and row levels. The table in Figure 1 shows an example table. The column ROW_CLASS contains the row classifications and the value in square brackets is the table classification.

[1]Pollyanna: "A blindly optimistic person" - Webster's College Dictionary

BEN.CARS [UNCLASSIFIED]

ROW_CLASS	MAKE	PRICE
UNCLASSIFIED	Nissan Micra	20000
CONFIDENTIAL	Hillman	104686
SECRET	Robin Reliant	12345

Figure 1. Example Polyinstantiating Database Table.

For a user to be able to detect the existence of the table, a prerequisite of being able to use it in any way, the session level must dominate the table classification.

To be able to detect a row and observe its contents, the session level must also dominate the ROW_CLASS. That is, the ROW_CLASS classifies the existence, classification and content of all fields in the row.

2.3 The Pollyanna Query Language Interface

The front end sits between the users' processes and the database (see figure 2). The program takes requests from the users and then filters the information to uphold the confidentiality constraints. The requests that the user makes are in a simple version of SQL which has additional constructs to handle classifications.

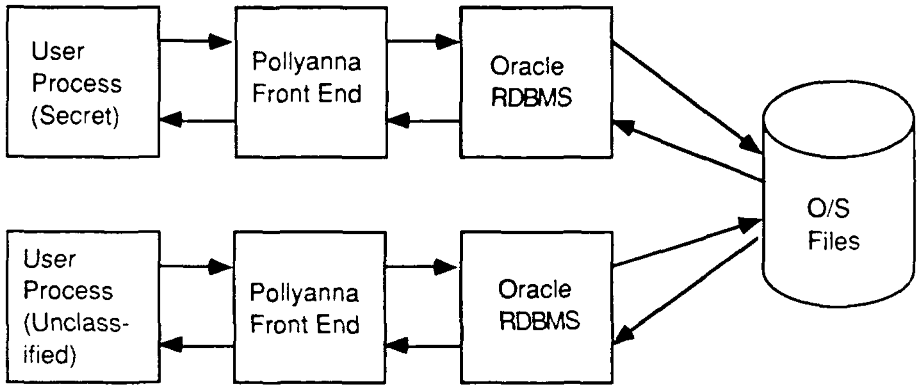


Figure 2. The Front End Architecture

2.3.1 SELECT

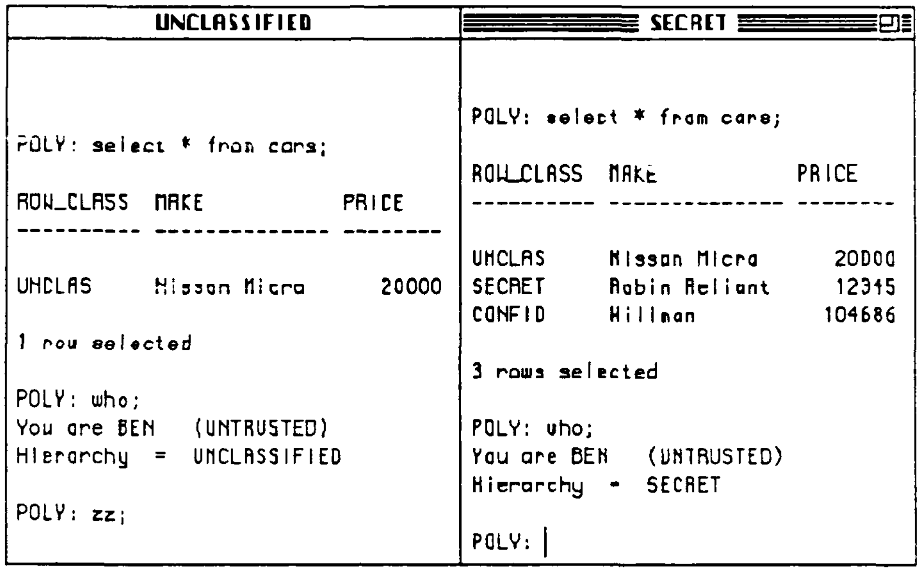
```
tableName ::= [ userIdent . ] tableIdent
colName ::= [ tableName . ] colIdent

factor ::= [ string | integer | colName ]
term ::= [ factor | factor [ + | - ] term ]
expression ::= [ term | term [ * | / ] expression | ( expression) ]
relation ::= [ expression [ = | <> ] expression ]
condition ::=   [ relation | relation [ AND | OR ] condition
                                    | NOT condition | ( condition ) ]

SELECT [ colName, ...... | * ] FROM tableName[ WHERE condition ]
```

A Select request returns values for each row which meets the criteria given in the where clause and whose row class is dominated by the session level.

An example is shown in figure 3. This is a screen copy of the Apple Macintosh front end implementation. The example shows two views of the same table (the table is the same as given in figure 1). The session level for each window is displayed on the window's title bar and also in the report from the 'who' command.


UNCLASSIFIED

```
POLY: select * from cars;

ROW_CLASS  MAKE            PRICE
---------- -------------- --------

UNCLAS     Nissan Micra      20000

1 row selected

POLY: who;
You are BEN    (UNTRUSTED)
Hierarchy  =  UNCLASSIFIED

POLY: zz;
```

SECRET

```
POLY: select * from cars;

ROW_CLASS  MAKE            PRICE
---------- -------------- --------

UNCLAS     Nissan Micra      20000
SECRET     Robin Reliant     12345
CONFID     Hillman          104686

3 rows selected

POLY: who;
You are BEN    (UNTRUSTED)
Hierarchy  =  SECRET

POLY: |
```

Figure 3. Sample output from two select requests to the Pollyanna RDBMS

2.3.2 INSERT

INSERT INTO tableName [(colIdent, ...)] VALUES ([string | integer], ...)

An insert adds a new row to the table, with the ROW_CLASS set to the session level. The user is not permitted to set the row class explicitly. The number of values to be inserted must be the same as the number of columns in the table (not counting the ROW_CLASS).

2.3.3 DELETE

DELETE FROM tableName [WHERE condition]

A delete removes all rows which satisfy the condition given in the where clause, and whose row class equals the session level.

2.3.4 UPDATE

UPDATE tableName SET colIdent = expression, ... [WHERE condition]

The expressions in the set clauses may not refer to any columns other than those being set. That is copying from one column to another is not allowed, but modifying a column based on its original values is allowed.

An update affects all rows which satisfy the condition given in the where clause, and whose row class is dominated by the session level. A copy is made of those affected rows whose row class is lower than the session level. The row class of the copy is changed to the session level and only this copy is updated. Those affected rows whose row class equals the session level are updated directly.

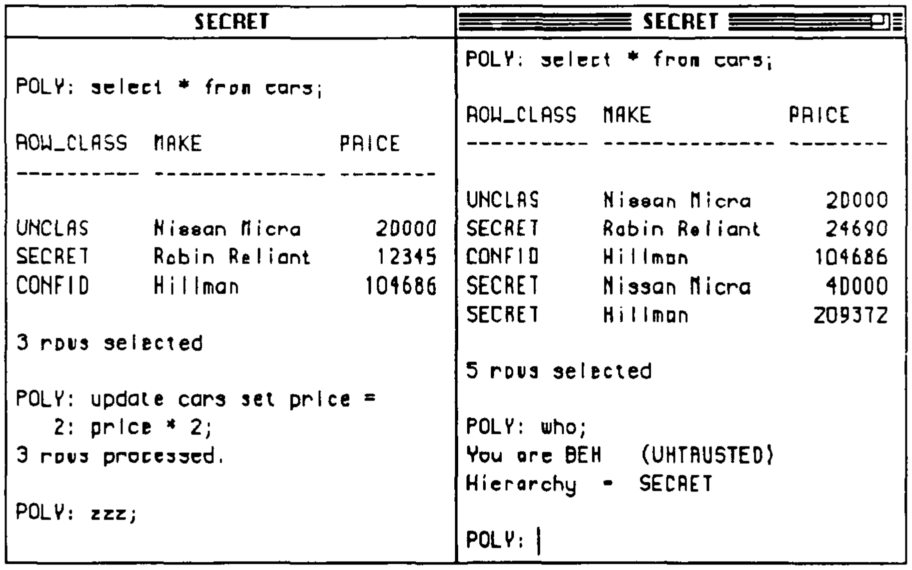
For example, take the case of a Secret user updating the table described in figure 1 with the query :-

```
UPDATE CARS SET PRICE = PRICE * 2;
```

Looking at the table 'cars' we see that the call will affect 3 rows. The classification of the row (UNCLASSIFIED, 'Nissan Micra', 20000) is lower than the session level. Therefore a copy of the original row will be retained and a new row will be inserted with the information (SECRET, 'Nissan Micra', 40000). A similar transformation will be made to the row (CONFIDENTIAL, 'Hillman', 104686). However the classification of the row (SECRET, 'Robin Reliant', 12345) equals the session level. Therefore the row is simply updated to become (SECRET, 'Robin Reliant', 24690).

The effect of this is to turn a table with three rows into a table with five. This is one of the strange effects of polyinstantiation - an update may increase the number of rows [5]. Figure 4 shows the output from the Macintosh front end before and after the update described in the example above. The left hand window shows a select made before the update and the actual call for the update. The right hand window shows a select made after the update and details of the session level for both windows.

It is not possible to update a field which is designated as being part of a unique key. This restriction is made because it is found in most polyinstantiating DBMSs.



```
POLY: select * from cars;

ROW_CLASS  MAKE            PRICE
---------- -------------- --------

UNCLAS     Nissan Micra      20000
SECRET     Robin Reliant     12345
CONFID     Hillman          104686

3 rows selected

POLY: update cars set price =
  2: price * 2;
3 rows processed.

POLY: zzz;
```

```
POLY: select * from cars;

ROW_CLASS  MAKE            PRICE
---------- -------------- --------

UNCLAS     Nissan Micra      20000
SECRET     Robin Reliant     24690
CONFID     Hillman          104686
SECRET     Nissan Micra      40000
SECRET     Hillman          209372

5 rows selected

POLY: who;
You are BEH   (UNTRUSTED)
Hierarchy  -  SECRET

POLY: |
```

Figure 4. Two windows showing the effect of an update in Pollyanna

2.3.5 CREATE TABLE

```
type ::= [ [ NUMBER | CHAR | VARCHAR ] [ ( integer ) ] | DATE | FLOAT ]

CREATE TABLE tableIdent ( colIdent type [ [ NOT ] NULL ], ...... )
```

A create table query creates a new table with the specified columns. An extra column, ROW_CLASS, is also created implicitly. The table classification of the new table is set equal to the session level. The owner of the new table is the user making the query.

2.3.6 CREATE UNIQUE INDEX

```
CREATE UNIQUE INDEX indexIdent ON tableName ( colIdent, ...... )
```

A unique index can be created on some specified set of columns. The ROW_CLASS column is implicitly added to this set.

2.3.7 Others
Other queries are as for standard SQL.

```
DROP TABLE tableName
CREATE INDEX indexIdent ON tableName ( colIdent, ...... )
DROP INDEX indexIdent
```

3. THE FRONT END IMPLEMENTATION OF POLLYANNA

3.1 The Front End's Internal Tables

For Pollyanna controls to be enforced, the classifications of tables and rows have to be stored. The table classes for all tables are held in a table called TABLE_DETAILS. Row classes are held in an additional column, called ROW_CLASS. Key fields are recorded in another table, TABLE_KEYS. This is needed to support the prohibition on updates of key fields.

These table are owned by a special Oracle user which represents the Pollyanna DBMS. This user does not correspond to any Pollyanna user and the tables are not directly accessible to any Pollyanna users. The tables must be modifiable by the front end on behalf of all Pollyanna users. To this end global access privileges are granted on the tables to the public user group.

Figure 5 gives an example of the two tables.

TABLE_DETAILS

T_NAME	CLASS	COL_NOS
BEN.CARS	0	2

TABLE_KEYS

K_NAME	K_OWNER	T_NAME	K_LIST
DRIVE	BEN	BEN.CARS	ROW_CLASS, MAKE

Figure 5. The tables needed for the Pollyanna front end

The TABLE_DETAILS table has three columns:

T_NAME
: The name of the table being created for the user.

CLASS
: The underlying representation of the table classification.

COL_NOS
: This contains the number of columns used in the table. However the column ROW_CLASS is not taken into account, meaning that there are actually COL_NOS + 1 columns in the underlying table.

Rows are inserted into TABLE_DETAILS during the creation of a new table and are deleted during the handling of a table drop. The entries are never updated.

The TABLE_KEYS table has four columns:

K_NAME
: The name of the unique key created for the user.

K_OWNER
: The name of the owner of the unique key.

T_NAME
: The name of the table on which the unique key is imposed.

K_LIST
: This column contains the list of all the columns used in the unique key. The column names are separated by commas.

Records are inserted into TABLE_KEYS during the creation of a new unique key and are deleted during the handling of a unique key drop. The entries are never updated.

Additional tables contain information regarding the names and clearances of the authorised Pollyanna users, and the conversions between textual classifications and their underlying representation.

3.2 Implementing Queries

A user can only make insert, select, update or delete queries on a table if their session level dominates the table's classification. This is checked by extracting the table class from the TABLE_DETAILS table. If the table does not exist or the session level does not dominate the table's classification, a "table does not exist" error is returned to the user.

3.2.1 SELECT

A select query is modified so that it also retrieves the ROW_CLASS column, in addition to any columns requested by the user. If the user requests all columns, by using a *, the ROW_CLASS is returned anyway. Those rows retrieved whose row class is dominated by the session level are passed on to the user. Any others are discarded.

As the ROW_CLASS had to be retrieved for filtering, it seemed sensible to display it. The hierarchical part of the row classification is shown, followed by an ellipsis if any caveats and categories are also included.

3.2.2 UPDATE

An update query in Pollyanna is implemented by sending three queries to Oracle. The first query updates all rows with the same ROW_CLASS as the present session level. This query is formed from the original by extending the where clause with:

```
UPDATE ...... WHERE (......) AND ROW_CLASS = session_level
```

The second query is a select that retrieves all rows meeting the condition of the where clause which have a row class not equal to the session level:

```
SELECT * FROM ...... WHERE (...... ) AND ROW_CLASS <> session_level
```

The front end then discards all rows not dominated by the session level. The remaining rows are modified as requested by the SET clause and the row class is changed to the session level. The new row is then inserted into the table.

    INSERT INTO ...... VALUES(session_level, ..., ...)

As the update is done in two stages, failure of the system at any point means that there could well be records that are now inconsistent and need to be returned to their former state. For simplicity, a simple "rollback" is used, even though this may well cancel the actions of previous queries. A more complex solution would involve the front end remembering what records have been modified and in what way, therefore allowing the front end to undo the modifications.

During an update, problems occur when the set of affected rows contains rows with the same values in the unique keys except for a difference in the ROW_CLASS, for example, the rows (SECRET, 'Beetle', 4300) and (UNCLASSIFIED, 'Beetle', 34322) where the unique key is on the first two columns.

If two or more such records are dominated by the present session level then the results of the update become ambiguous. The only sensible solution to this problem seems to be to return an error to the user.

The problem can only be solved by the user rewriting the update request to select only 1 of the clashing records. Unfortunately this may not always be possible and the update request may have to be made as two distinct queries.

3.2.3 INSERT
As inserted rows are given the same ROW_CLASS as the session level then it is the job of the front end to make sure that the user doesn't try to define a different value for it. If the user specifies the ordering of the columns to be set then the front end adds the column ROW_CLASS to the beginning of the column name list and the session level to the front of the list of values. If the order is not specified, the front end just adds the session level to the beginning of the list of inserted values.

    INSERT INTO ...... (ROW_CLASS, ...) VALUES (session_level, ..., ...)
    INSERT INTO ......VALUES (session_level, ..., ...)

3.2.4 DELETE
For deletes the front end adds to the users where clause the extra condition that the row class must equal the session level:

    DELETE FROM ...... WHERE (......) AND ROW_CLASS = session_level

4. EXAMPLE

To illustrate the problems of polyinstantiation, consider a simple example database which records the destinations of aircraft flights. There is just one table, called flights, which is originally empty. The table has two columns, Fno and Dest, plus the 'virtual' column RowClass.

RowClass	Fno	Dest

Now suppose a Secret user inserts a row to show that flight 100 is heading for Ascension on a Secret mission. The new row is labelled Secret and so will be invisible to any users with lower clearances. No problems so far.

RowClass	Fno	Dest
Secret	100	Ascension

If, however, an Unclassified user wants to send an aircraft to Paris, the problems begin The user would look at the table to see which aircraft are busy with the following query:

    SELECT Fno FROM flights

This would retrieve no rows, because the table only contains a row which is hidden from the user. Therefore the Unclassified user may conclude that flight 100 is idle and choose this for the flight to Paris:

    INSERT INTO flights VALUES(100, 'Paris')

This results in the table recording two destinations for flight number 100. The Unclassified user sees just one row, and so is not confused, but the Secret user sees both.

RowClass	Fno	Dest
Secret	100	Ascension
Unclass	100	Paris

Now suppose the Unclassified user chooses another flight. This time a number other than 100 must be chosen, because it is seen to be in use. Suppose it is decided that flight number 200 is to go to Rome:

RowClass	Fno	Dest
Secret	100	Ascension
Unclass	100	Paris
Unclass	200	Rome

The Secret user can see that flight 200 is assigned to Rome, but suppose that emergency arises and flight 200 must be diverted on a Secret mission to Gander:

    UPDATE flights SET Dest = 'Gander' WHERE Fno = 200

The row for flight 200 is Unclassified, so the Secret user cannot update it directly without some information flowing to the Unclassified user. Therefore, Pollyanna 'polyinstantiates' the row, effectively turning the update into an insert. This results in a new row which shows flight number [illegible] heading for Gander.

RowClass	Fno	Dest
Secret	100	Ascension
Unclass	100	Paris
Unclass	200	Rome
Secret	200	Gander

Now flight 200 has two destinations which is bound to cause trouble.

The problems occur because Pollyanna allows just one classification for everything about a row, including its existence, its classification and the values of all its fields. The key to the success of the SWORD DBMS is that it does provide finer grained classifications, which not only allow individual fields to be separately classified, but also the fields to be classified higher than the existence of the row.

5. CONCLUSIONS

Pollyanna was designed to be used as a demonstration of the polyinstantiation approach and to be used in comparative demonstrations with the SWORD system.

The design of Pollyanna has highlighted some problems with the effects of polyinstantiation on the meaning of update queries. In Pollyanna it was decided to reject update queries that result in duplicate keys. Other solutions are possible, including only updating the most highly classified version of a row. Future work could provide these alternatives as an option and allow further evaluation and comparison of the different approaches.

Further extensions are possible to relax the rather strong constraints on set lists in update queries. These constraints were imposed to ease the implementation but are not inherent in the design of Pollyanna. Also improvements could be made to the error recovery code, so that an erroneous query does not cause a rollback of all previous work in the transaction.

6. REFERENCES

[1] D.E.Denning, T.F.Lunt, R.R.Schell, M.Heckman, W.Shockley, A Multilevel Relational Data Model, Proceedings 1987 IEEE Symposium on Security and Privacy, April 27-29, Oakland, California, pp220-234.

[2] A.Wood, The SWORD Model of Multilevel Secure Databases, Royal Signals and Radar Establishment Report 90008, June 1990.

[3] S.R.Lewis, The Front End Approach to Database Security, Proceedings of the seventh international IFIP TC11 Conference on Information Security, Brighton, UK, 15-17 May 1991.

[4] Simon Wiseman, On the Problem of Security in Data Bases, Procs. IFIP WG11.3 Workshop of Database Security, Monterey CA, September 1989.

[5] S.Jajodia & R.Sandhu, Polinstantiation Integrity in Multilevel Relations, Procs Symp. Security and Privacy, Oakland, CA, May 1990, pp104-115.

REPORT DOCUMENTATION PAGE DRIC Reference Number (if known) ....

Overall security classification of sheet ..................................UNCLASSIFIED..

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification eg (R), (C) or (S).

Originators Reference/Report No. MEMO 4540	Month NOVEMBER	Year 1991

Originators Name and Location RSRE, St Andrews Road Malvern, Worcs WR14 3PS
Monitoring Agency Name and Location
Title A FRONT END IMPLEMENTATION OF THE POLLYANNA SECURE DBMS

Report Security Classification UNCLASSIFIED	Title Classification (U, R, C or S) U

Foreign Language Title (in the case of translations)
Conference Details

Agency Reference	Contract Number and Period
Project Number	Other References
Authors HUTCHINSON, G R	Pagination and Ref 9

Abstract

Pollyanna is a simple secure DBMS which utilises polyinstantiation. It has been designed and implemented to illustrate the problems which may be encountered when applications are built on top of a polyinstantiating DBMS. The implementation comprises of a front end to a standard commercial RDBMS. The front end performs query modification and result filtering to achieve security. This document describes Pollyanna, the front end software and some of the problems which arose during the design of the system.

Abstract Classification (U,R,C or S)
U

Descriptors

Distribution Statement (Enter any limitations on the distribution of the document)

UNLIMITED

S80/48